Rejection Sampling

Math 365 Prof Leise

March 22, 2022

In this lab we will look at rejection sampling and the Metropolis-Hastings algorithm for Markov chain Monte Carlo simulations.

Submit the pdf knit from your completed lab to Gradescope.

Monte Carlo Integration

We start with the basic idea behind Monte Carlo simulations. Suppose we want to approximate π. Imagine throwing lots of darts at a 1 meter by 1 meter square board. Draw a quarter circle of radius 1 on the board. If your aim is uniformly random so that you are equally likely to hit any point on the square, then after throwing many many darts, you would expect the proportion of darts landing inside the quarter circle to be equal to its area, $\pi/4$. The more darts you throw, the better the approximation of π you will tend to get.

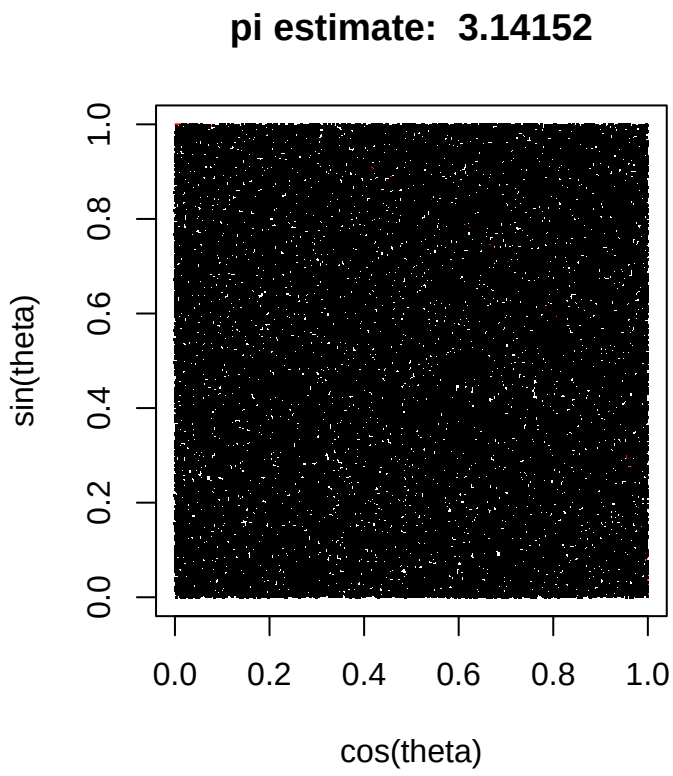
Exercise 1

Simulate this experiment in R by generating many points on the unit square, then counting how many landed in the unit circle. Multiply by 4 to obtain an estimate of π. How good is your approximation if you throw 100 darts? 1,000 darts? 10,000 darts? 100,000 darts?

Ans: The approximation gets closer and closer to the actual value as we increase the number of trials in form of the number of darts.

```
Ndarts <-100000
x <- runif(Ndarts) # uniform on [0,1]
y <- runif(Ndarts)
r2 <- x^2+y^2
estimate <- sum(r2<=1)/Ndarts*4 # estimate of pi

theta <- seq(0, pi/2,.01) # to draw circle
par(pty="s") # makes a square plot
plot(cos(theta),sin(theta),col="red",type="l",xlim=c(0,1),ylim=c(0,1),main=paste("pi estimate: ",estima
points(x,y,pch=".") # darts on board
```




```
par(pty="m") # resets so plots below have optimal aspect ratio
```

Rejection Sampling

Now suppose you want to generate random numbers from an unusual probability density function, for example, something strange like $f(x) = \frac{x(1-x)e^x}{3-e}$ on $[0,1]$. We want a method that uses random numbers from a density $h(x)$ we know how to calculate, e.g., uniformly distributed random numbers with $h(x) = 1$ on $[0,1]$, and outputs random numbers according to the desired density $f(x)$. Rejection sampling is one way to accomplish this. First find a constant c such that $f(x) \leq ch(x)$ for all $x \in [0,1]$, then follow these steps:

$c = 1.5$ (i) Generate a random number x with the density $h(x)$.

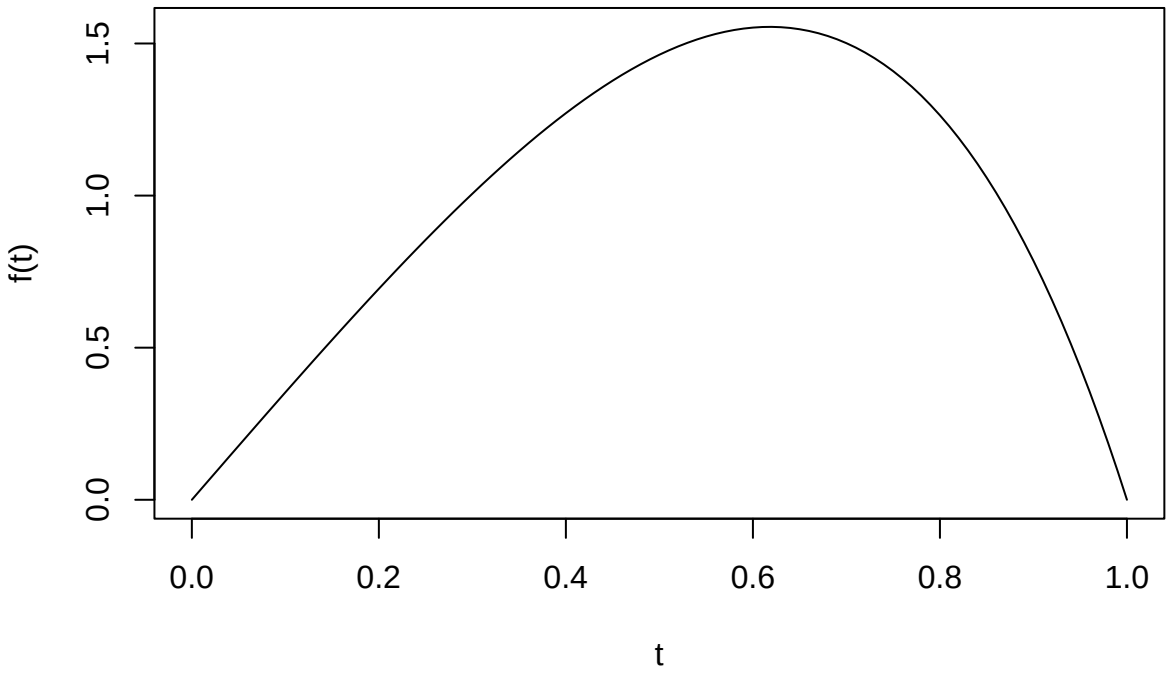
(ii) Generate a uniformly distributed random number u from the interval $[0,1]$. If $u \leq \frac{f(x)}{ch(x)}$, then output x; otherwise reject x and return to step 1.

This procedure uses the same underlying idea as the darts example above: Throw many darts at a board that is c high and with base $[0,1]$ on which we have drawn our desired density $f(x)$. Reject the darts that land above the curve, and accept those that land under it. There will be more darts near values of x for which $f(x)$ is large than where $f(x)$ is small, so with enough darts you will approximate the density function.

Exercise 2

Use rejection sampling to generate a set of random numbers according to the density function $f(x) = \frac{x(1-x)e^x}{3-e}$ on $[0,1]$.

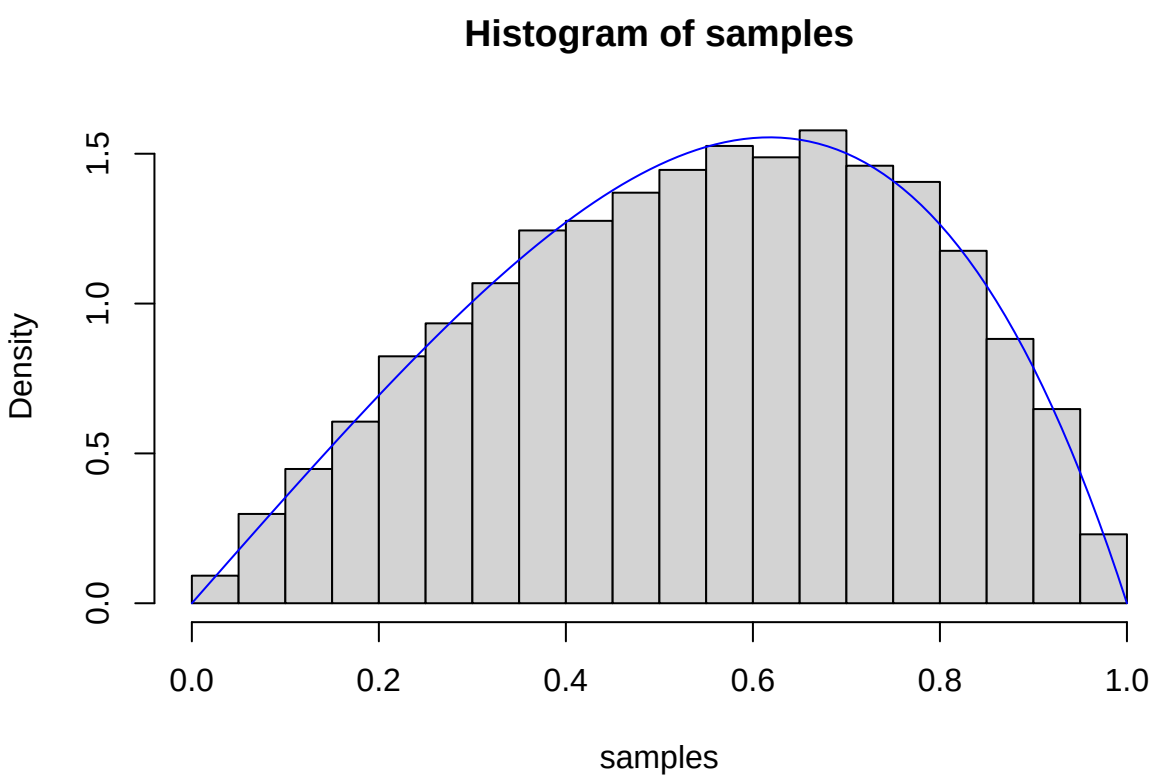
```r
f <- function(x) x*(1-x)*exp(x)/(3-exp(1))
t <- seq(0,1,0.01)
plot(t,f(t),type="l") # graph of the desired density function
```




```r
rdist <- function(c) { # choose c with f(x)<=c for all x
  for (i in 1:1000) {
    x <- runif(1,0,1) # step 1
    u <- runif(1,0,1) # step 2
    ifelse(c*u<=f(x),return(x),-1)
  }
  return(-1) # if tried 1000 times and rejected all
}
```

The command rdist(c), where you put an appropriate value in for c based on the graph of $f(x)$, will generate a single random number. The code below generates 10,000 such random numbers and plots a histogram using freq=FALSE (so it plots the density) along with a plot of $f(x)$ using the points command in R to check that the histogram indeed closely matches the desired density function.

```r
c <- 1.5 # enter a value of c that is any upper bound on f(x)
Nsamples <- 10000
samples <- matrix(0,Nsamples,1)
for (k in 1:Nsamples) samples[k] <- rdist(c)
hist(samples,freq=FALSE)
points(t,f(t),type="l",col="blue")
```



Metropolis-Hastings Algorithm

Suppose you want to generate a discrete-time Markov chain whose invariant distribution is a given vector π. The Metropolis-Hastings algorithms takes any Markov chain and uses it to generate the desired Markov chain.

To illustrate the algorithm, let $S = \{0, 1, 2, 3\}$. Let the desired invariant probability vector be $\pi =$ [0.1 0.2 0.3 0.4]. That is, we want the chain to be at state 0 10% of the time, at state 1 20% of the time, etc. Take P to be the transition matrix of any convenient Markov chain, for example, random walk on the circle with $p = \frac{1}{2}$. To generate a new Markov chain, we need to specify the transitions. Suppose $X_m = i$. The next state X_{m+1} is determined by a two-step procedure:

(i) Choose a proposal state j using the transition probabilities T_{ij}. For the random walk on a circle, the proposal state j will be either $i - 1$ or $i + 1$ (mod 4).

(ii) Decide whether or not to accept the proposal state, according to the acceptance function

$$a(i, j) = \frac{\pi(j)T_{ji}}{\pi(i)T_{ij}}.$$

Generate a uniformly distributed random number u from the interval $[0, 1]$. The next state X_{m+1} of the chain is j if $u \leq a(i, j)$ and i if $u > a(i, j)$.

In this simple case, since $p = \frac{1}{2}$, $T_{ji} = T_{ij}$, so the acceptance function simplifies to $a(i, j) = \frac{\pi(j)}{\pi(i)}$. Hence the next step will be j if $u \leq \frac{\pi(j)}{\pi(i)}$; otherwise it is again i.

```r
pi <- c(0.1, 0.2, 0.3, 0.4)

mcmc <- function(n) { # n=number of steps to take
  currentstate <- 0 # initial state X0=0
  for (i in 1:n) {
    proposal <- (currentstate + sample(c(-1,1),1)) %% 4 # mod 4
    accept <- pi[proposal+1]/pi[currentstate+1]
    u <- runif(1,0,1)
    if (u < accept) currentstate <- proposal  }
  currentstate
}
```

The mcmc(n) function outputs X_n. We can choose, for example, $n = 100$ and examine the distribution of X_{100} from 10,000 simulations:

```r
trials <- 10000
simlist <- replicate(trials,mcmc(100))
table(simlist)/trials # proportion in each state 0,1,2,3
```

```
## simlist
##      0      1      2      3
## 0.1070 0.1957 0.2950 0.4023
```

Exercise 3

How well did the MCMC generated distribution mimic the desired distribution π?

Ans: I think the MCMC generated distribution modeled the desired distribution pretty well as we can see that each of the elements in the simlist on average is close to the corresponding elements in the pi_bar vector which was set as our target.